CSE 464: Software Quality Assurance and Testing
Course Project Part-3

Vajravel Conjeevaram Manigandan
1225428152
vconjeev@asu.edu

Introduction:

This is a Java based application that can parse graphs in DOT format, manipulate graphs (e.g., add nodes, add edges), and output graphs in DOT format and graphics (e.g., jpg and png).

This project was built using **IntelliJ** as the IDE, **Java 21** as the Java Development Kit (JDK), **Maven** to manage dependencies, and the **graphviz-java** was used to parse the DOT file into objects in java and vice versa. Classes such as Graph Manager, Graph, Node, and Edge were implemented to perform operations such as coding the features, creating a graph, adding nodes or edges on the given graph. Also the Path class was added to implement new features such as Breadth-first-search(BFS) and Depth-first-search(DFS). Furthermore, new classes have been created to implement another type of graph searcher algorithm called Random Walk Search as well as implementing the Template and Strategy design patterns.

NOTE: ***If there are any dependency issues, please resolve them using mvn install in the command line terminal or resolve them using maven in IntelliJ.***

Access the project:

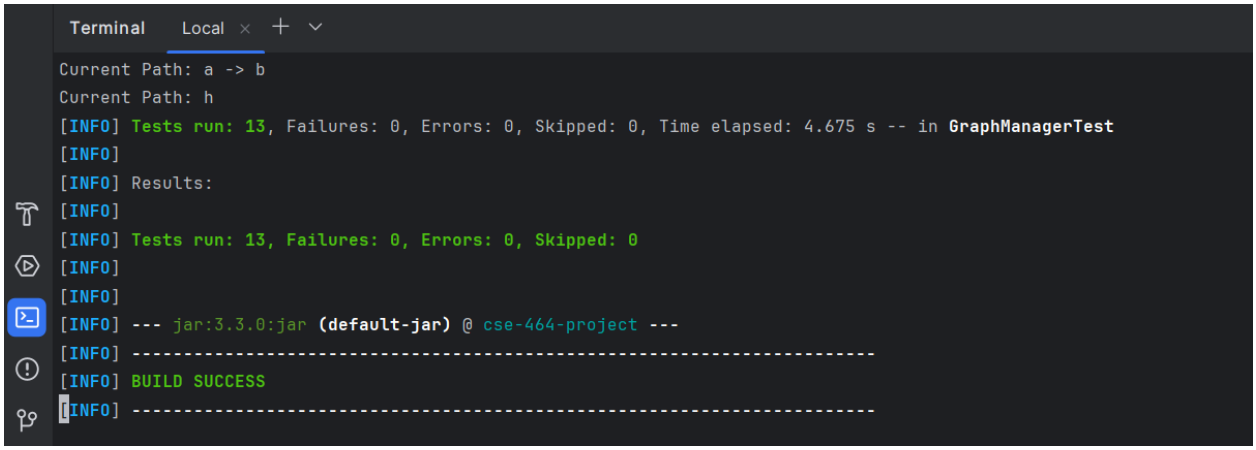
Every step of the project has been tracked using Github, and can be accessed via the link:
https://github.com/v-Cm/CSE-464-2023-vconjeev

Run the project:

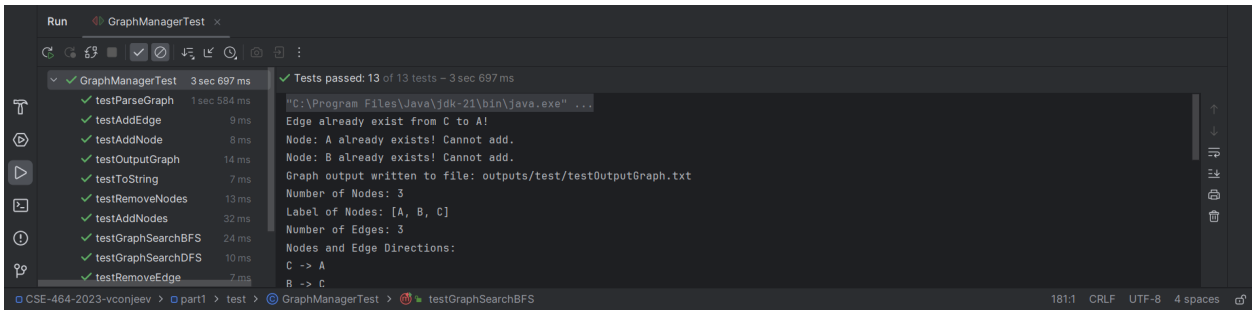
Clone or download the project folder to your system. If you are using an IDE (IntelliJ) to access the project, you can simply run the **Application.java** file in the **/part1/src/main/java/** path. Or, if you are using the CLI/terminal you can type **mvn package** to compile and run the project.

Test the project:

The application has several features in it (which will be described in detail in the following section) and the expected outputs for each features have been provided either in the **GraphManagerTest.java** file in the /part1/test/ path, or as files in the /**part1/outputs/expected** directory. Again, you can run **mvn test** or **mvn package** on the terminal,



or run the testcases in the **GraphManagerTest.java** file manually.



Five new testcases have been added to check all the features added in this part of the project.

About the features:

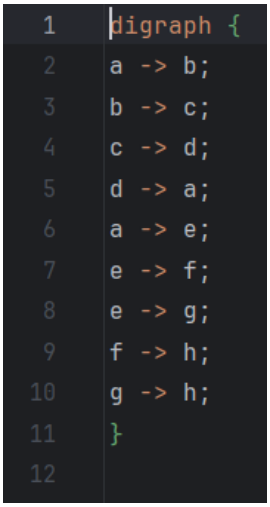
All the features required in part 3 are added as Classes named “BFSStrategy.java”, “DFSStrategy.java”, “RandomWalkStrategy.java”, “RandomWalk.java”, “GraphSearchTemplate.java”, “GraphSearchStrategy.java”, “GraphSearchContext.java” which are in the folder: part1/src/main/java.

Input:

1. **testInput.dot** (which can be edited however you want).

```
digraph {
    A -> B;
    B -> C;
    C -> A;
}
```

2. **Input2.dot**



Refactorings: .

- extract variable on "->" in Graph.java for reusability - Github commit link.
- extract method on findPathUsingBFS in Graph.java for improved readability - Github commit link.
- extract method removeAssociatedEdges from removeNode in Graph.java for improved readability- Github commit link.
- extract variable on createEdge in Graph.java for improved readability and reusability - Github commit link.
- extract method in Application.java for reusabilty and prevent duplication- Github commit link.

Template Design Pattern: Created a new **GraphSearchTemplate** class which is the template and has an abstract method **exploreNeighbors(string, string)** which is implemented in BFS, DFS, and RandomWalk classes.
Github commit link.

Strategy Design Pattern: Created a new **GraphSearchStrategy** interface which is the strategy class. The **GraphSearch** API in GraphManager.java is the client code and **GraphSearchContext** is the context class. The **BFSStrategy**, **DFSStrategy**, and **RandomWalkStrategy** are the concrete classes that implement the interface.
Github commit link.

BFS/DFS:

1. To run this algorithm, execute Application.java or the test file. The performGraphSearch(GraphManager, Node, Node, Algorithm) would invoke the GraphSearch API
2. This would call the Path GraphSearch(String src, String dst, Algorithm algo) API which would pass on the corresponding strategy to the context class, and thus invoking the strategy concrete class. Here algo can be Algorithm.BFS or Algorithm.DFS.
3. Finally, the search method is executed which contains the actual BFS/DFS algorithms and will return the path.If no path exists, it will return null.

```
10      performGraphSearch(graphManager, new Node( name: "A"), new Node( name: "A"), Algorithm.BFS);
11      performGraphSearch(graphManager, new Node( name: "B"), new Node( name: "A"), Algorithm.BFS);
12      performGraphSearch(graphManager, new Node( name: "A"), new Node( name: "A"), Algorithm.DFS);
13      performGraphSearch(graphManager, new Node( name: "B"), new Node( name: "A"), Algorithm.DFS);
```

Invoking the performGraphSearch API from Application.java

```
The path:A
The path:B -> C -> A
The path:A
The path:B -> C -> A
```

Output of BFS and DFS

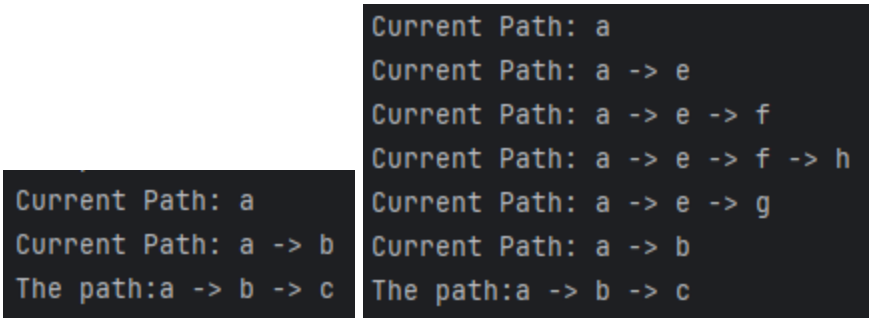
Random Walk Search:

[Github commit link]

1. Just like BFS and DFS, to run this algorithm, execute Application.java or the test file. The performGraphSearch(GraphManager, Node, Node, Algorithm) would invoke the GraphSearch API and everything that follows is exactly the same as above.
2. The Random Walk Search algorithm follows the very same logic as DFS, but, the only difference is that, the Random Walk algorithm chooses the next neighbor to travel down towards randomly during each run of the exact same graph.
3. Finally, the search method is executed which contains the actual Random Walk Search algorithm and will return a random path connecting the source and destination. The algorithm also prints the path travelled so far during each iteration. If no path exists, it will return null.

```
14      performGraphSearch(graphManager1, new Node( name: "a"), new Node( name: "c"), Algorithm.RANDOMWALK);
```

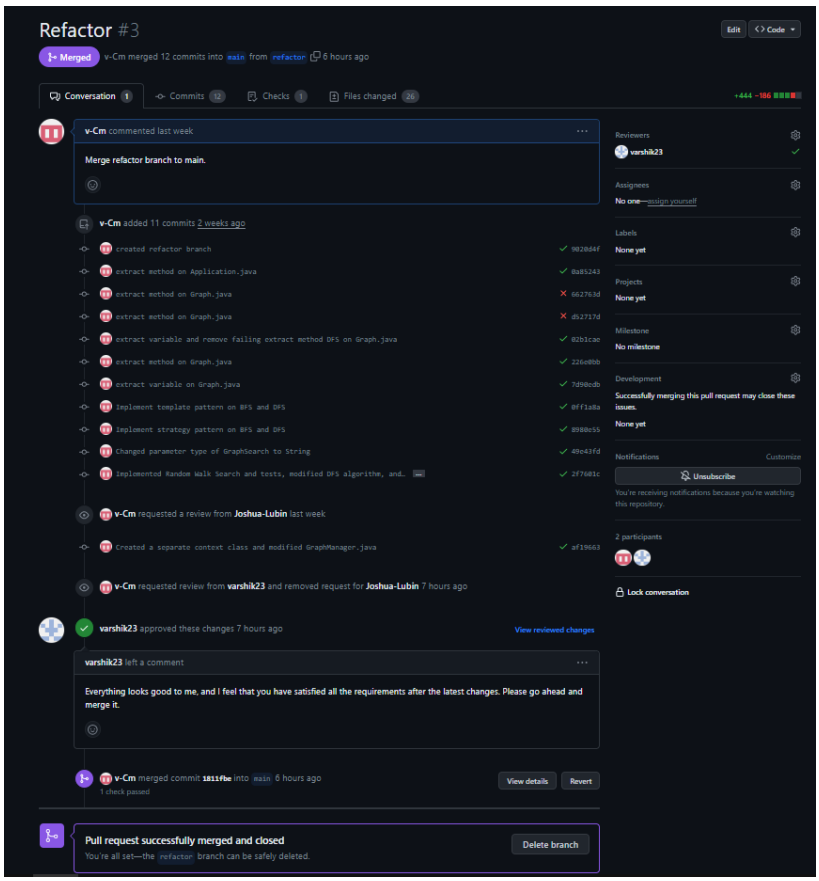
Invoking the performGraphSearch API from Application.java



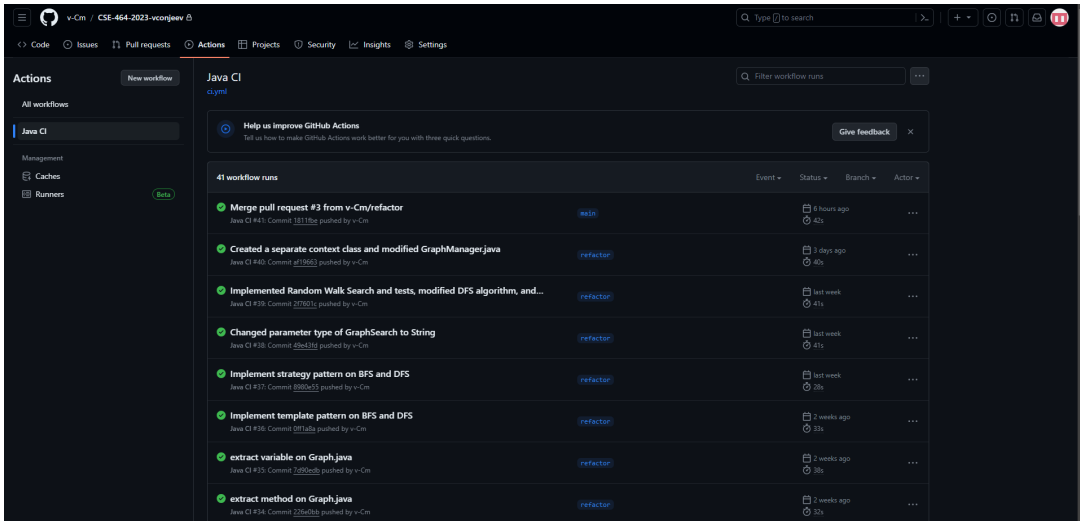
Multiple outputs of the API calls above to show randomness.

Pull request: I created the pull request to merge the refractor branch into the main branch. Firstly, I made 5 code refactors. Then, I implemented the template and strategy design patterns. Now, I implemented DFS, BFS, and Random Walk algorithms using this GraphSearchTemplate abstract class. Once the algorithms are implemented using the template pattern, The algorithm is made available to choose dynamically by implementing the strategy design pattern. The GraphSearchStrategy interface is the strategy class and I have implemented separate strategy classes for DFS, BFS, and Random Walk. The Algorithm enum contains DFS, BFS, RANDOMWALK, and INVALID as options. INVALID is used for testing purposes.

- Merge refactor to main link



Github workflow link:



Github commits

Github main branch commit log

Github refactor branch commit log